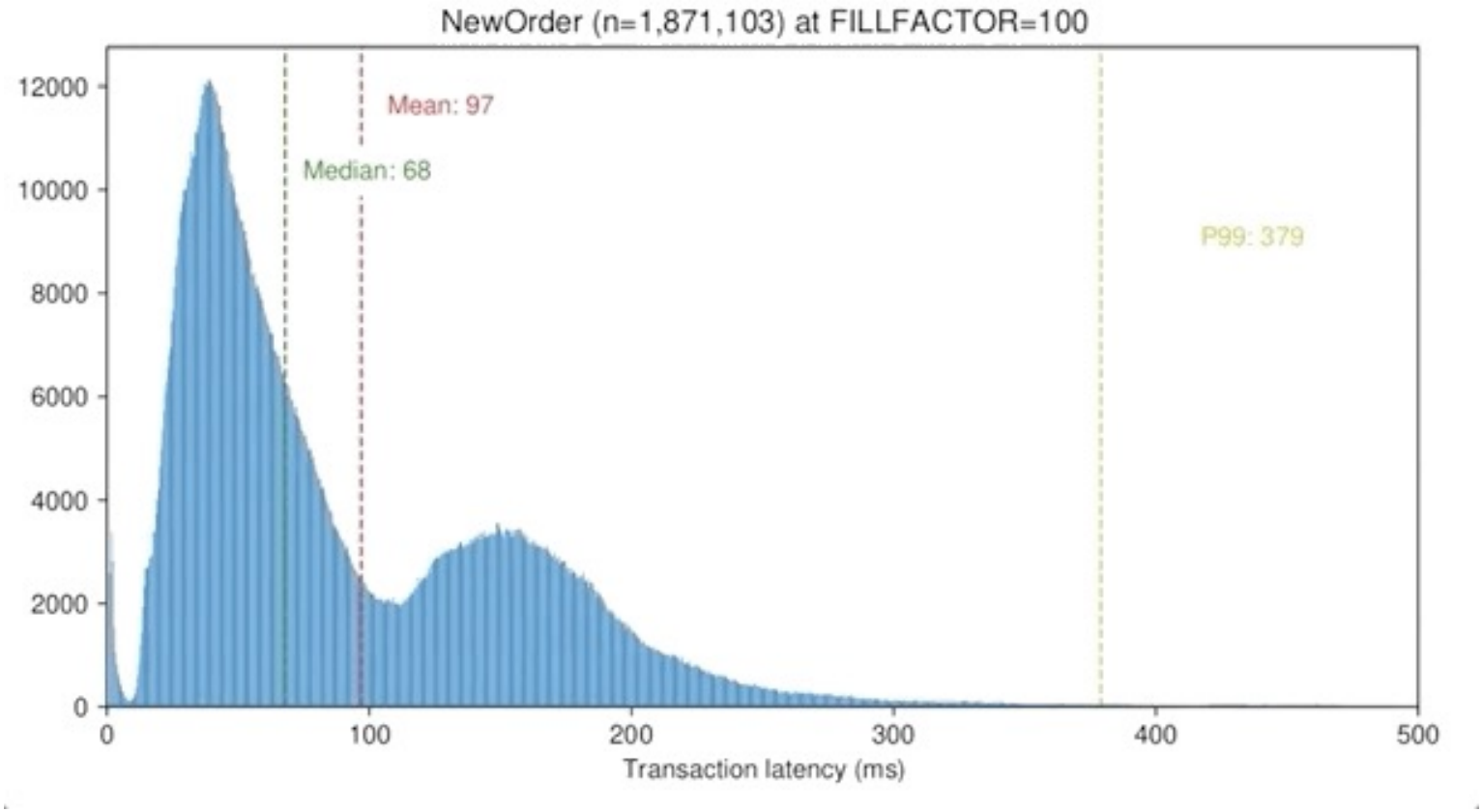
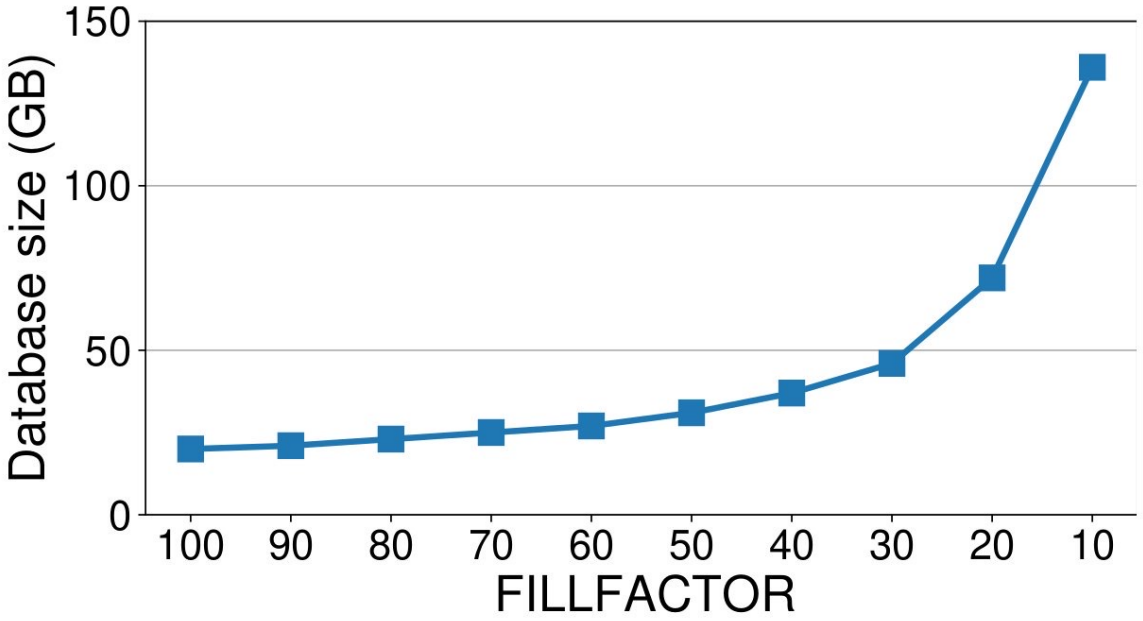
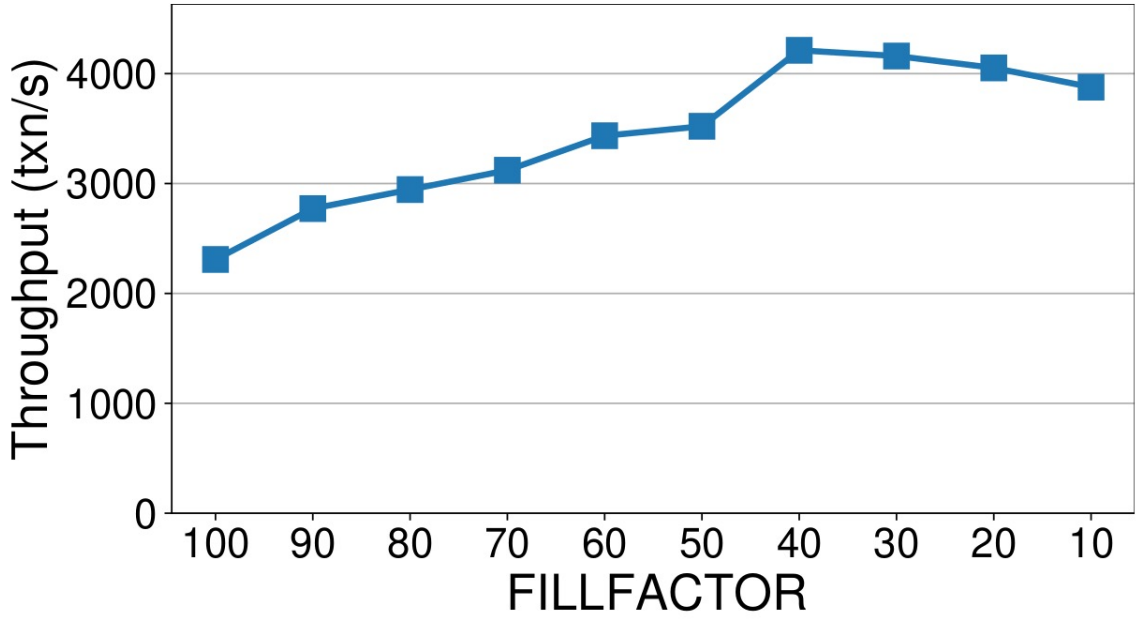
Matt Butrovich
@butro

We've been playing with @PostgreSQL's fillfactor for tables. This is a hard setting to get right because it depends on workload patterns. This graph shows TPC-C NewOrder latency as you change fillfactor on 5 tables. p99 gets tighter. Not sure of bimodal pattern at fillfactor=100.




Database size (GB)
150
100
50
0
100 90 80 70 60 50 40 30 20 10
FILLFACTOR


Throughput (txn/s)
4000
3000
2000
1000
0
100 90 80 70 60 50 40 30 20 10
FILLFACTOR

United States Department of Justice

THE UNITED STATES ATTORNEY'S OFFICE

SOUTHERN DISTRICT of NEW YORK

FOR IMMEDIATE RELEASE

Monday, November 7, 2022

U.S. Attorney Announces Historic $3.36 Billion Cryptocurrency Seizure And Conviction In Connection With Silk Road Dark Web Fraud

In November 2021, Law Enforcement Seized Over 50,676 Bitcoin Hidden in Devices in Defendant JAMES ZHONG's Home; ZHONG Has Now Pled Guilty to Unlawfully Obtaining that Bitcoin From the Silk Road Dark Web in 2012

https://www.justice.gov/usao-sdny/pr/us-attorney-announces-historic-336-billion-cryptocurrency-seizure-and-conviction

United States Department of Justice

THE UNITED STATES ATTORNEY'S OFFICE

SOUTHERN DISTRICT of NEW YORK

ZHONG funded the Fraud Accounts with an initial deposit of between 200 and 2,000 Bitcoin. After the initial deposit, ZHONG then quickly executed a series of withdrawals. Through his scheme to defraud, ZHONG was able to withdraw many times more Bitcoin out of Silk Road than he had deposited in the first instance. As an example, on September 19, 2012, ZHONG deposited 500 Bitcoin into a Silk Road wallet. Less than five seconds after making the initial deposit, ZHONG executed five withdrawals of 500 Bitcoin in rapid succession — *i.e.*, within the same second — resulting in a net gain of 2,000 Bitcoin. As another example, a different Fraud Account made a single deposit and over 50 Bitcoin withdrawals before the account ceased its activity. ZHONG moved this Bitcoin out of Silk Road and, in a matter of days, consolidated them into two high-value amounts.

https://www.justice.gov/usao-sdny/pr/us-attorney-announces-historic-336-billion-cryptocurrency-seizure-and-conviction

Administrivia

Problem generators

- https://w4111.github.io/join.html
- https://w4111.github.io/concurrency.html

Project 1 Part 3 extended to next Tuesday
(see ed post)

Project 1 Part 3 meetings!

Transactions, Concurrency, Recovery

Eugene Wu

Overview

Why do we want transactions?

What guarantees do we want from transactions?

Why Transactions?

Concurrency (for performance)

- N clients, no concurrency
  - 1st client runs fast
  - 2nd client waits a bit
  - 3rd client waits a bit longer
  - Nth client walks away
- N clients, concurrency
  - client 1 runs x += y
  - client 2 runs x -= y
  - what happens?

Can we prevent stepping on toes? *Isolation*

```
User 1
x += y
a1 = read(x)
b1 = read(y)
store(a1 + b1)

User 2
x -= y
a2 = read(x)
b2 = read(y)
store(a2 - b2)
```

Good Execution Order

```
a1 = read(x)          // x=0
b1 = read(y)          // y=1
store(a1 + b1, x)     // 1→x
a2 = read(x)          // x=1
b2 = read(y)          // y=1
store(a2 - b2, x)     // 0→x
```

```
result:
x=0
```

```
User 1
x += y
a1 = read(x)
b1 = read(y)
store(a1 + b1)

User 2
x -= y
a2 = read(x)
b2 = read(y)
store(a2 – b2)
```

Bad Execution Order

```
a1 = read(x)          // x=0
a2 = read(x)          // x=0
b2 = read(y)          // y=1
store(a2 – b2, x)     // -1→x
b1 = read(y)          // y=1
store(a1 + b1, x)     // 1→x
```

result:
x=1

Why Transactions?

What about 1 client, no concurrency?

- Client runs big update query

  ```
  update set x += y
  ```

- Power goes out
- What is the state of the database?

Why Transactions?

What about 1 client, no concurrency?

- Client runs big update query
  - update set x += y
- Aborts the query (e.g., ctrl-c)
- What is the state of the database?

If an abort happens, can the database recover to something sensible? *Atomicity, Durability*

What does "sensible" mean?

Transactions = r/w over objects

Transaction: a sequence of actions

- action = read object, write object, commit, abort
- API between app semantics and DBMS's view

From app/user' point of view, the transaction (and its effects) are only "correct" once the DBMS has told the app/user that the transaction is COMMITed

Transactions = r/w over objects

Transaction: a sequence of actions

- action = read object, write object, commit, abort
- API between app semantics and DBMS's view

User's view

T1: begin	A=A+100	B=B-100	END
T2: begin	A=1.5*A	A=1.5*B	END

DBMS's logical view

T1: begin	r(A) w(A)	r(B) w(B)	END
T2: begin	r(A) w(A)	r(B) w(A)	END

ACID: Transaction Guarantees

Atomicity

users never see in-between xact state.
only see a xact's effects once it's commited
as if a xact runs instantaneously

Consistency

database always satisfies ICs.
xacts move from valid database to valid database

Isolation:

from xact's point of view, it's the only xact running

Durability:

if xact commits, its effects *must persist*

Concepts

Concurrency Control (CC)

techniques to ensure correct results when running transactions concurrently

what does this mean?

Recovery

On crash or abort, how to get back to a consistent (correct) state?

The two are intertwined! The CC mechanism dictates the complexity of recovery!

What Does Correct Execution Mean?

Serializability

Regardless of the interleaving of operations, end result same as a serial ordering (no concurrency)

Schedule

One specific interleaving of the operations

T1: R(A) R(B) W(D) COMMIT

How a Schedule Works

```
T1: A += 1
T2: A -= 1

Before T1 and T2, A is 0

   T1: R(A)       W(A,1) COMMIT
   T2:      R(A)                  W(A,-1) COMMIT
State: A=0  A=0    A=1    A=1     A=-1      A=-1
                uncommitted committed uncommitted committed
```

How a Schedule Works

```
T1: A += 1
T2: A -= 1

Before T1 and T2, A is 0

   T1: R(A)       W(A)   COMMIT
   T2:      R(A)                  W(A)     COMMIT
State: A=0  A=0    A       A       A          A
                uncommitted committed uncommitted    committed
```

The value that is written doesn't matter

Serial Schedules

Logical xacts

T1: r(A) w(A) r(B) w(B)

T2: r(A) w(A) r(B) w(B)

No concurrency (serial 1)

T1: r(A) w(A) r(B) w(B)

T2: r(A) w(A) r(B) w(B)

No concurrency (serial 2)

T1: r(A) w(A) r(B) w(B)

T2: r(A) w(A) r(B) w(B)

Are serial 1 and serial 2 equivalent?

More Example Schedules

Logical xacts

T1: r(A) w(A) r(A) w(B)

T2: r(A) w(A) r(B) w(B)

Concurrency (not equivalent to any serial schedule)

T1: r(A) w(A) r(A) w(B)

T2: r(A) w(A) r(B) w(B)

Concurrency (same as serial 1!)

T1: r(A) w(A) r(A) w(B)

T2: r(A) w(A) r(B) w(B)

Important Concepts

Serial schedule

single threaded model. no concurrency.

each xact finishes (commits or aborts) before next xact runs

Equivalent schedule

the database state same at end of both schedules

Serializable schedule (gold standard)

equivalent to a serial schedule

These are just definitions.

How to *ensure* that schedules are serializable?

SQL → R/W Operations

```
UPDATE   accounts
SET      bal = bal + 1000
WHERE    bal > 1M
```

Read all balances for every tuple

Update those with balances > 1000

Does the access method matter?

YES!

Tuples(objects) read depend on access method

R/W Operations Depend On Access Paths

```
UPDATE   accounts
SET      bal = bal + 1000
WHERE    id = 123
```

If 1000 tuples in accounts, how many tuples are read:

If no indexes?

If index on bal?

If hash index on id?

if B+-tree index on id?

R/W Operations Depend On Access Paths

```
UPDATE   accounts
SET      bal = bal + 1000
WHERE    id = 123
```

If 1000 tuples in accounts, how many tuples are read:

If no indexes?	1000 tuples
If index on bal?	1000 tuples
If hash index on id?	# tuples in hash bucket
if B+-tree index on id?	# tuples in a page

NonSerializable Schedule→Anomalies

Reading in-between (uncommitted) data

T1: R(A) W(A) R(B) W(B) abort

T2: R(A) W(A) commit

WR conflict or dirty reads

Reading same data gets different values

T1: R(A) R(A) W(A) commit

T2: R(A) W(A) commit

RW conflict or unrepeatable reads

NonSerializable Schedule→Anomalies

Stepping on someone else's writes

WW conflict or lost writes

Note: all anomalies involve writing to data that is read/written to.

If we track our writes, maybe can prevent anomalies

Conflict Serializability

cheaply prevent non-serializable scheds

Over-conservative: some serializable schedules disallowed.

Intuition: if xacts don't touch the same records, should be OK.

Conflict Serializability

What is a conflict?

For 2 operations, if run in different order, get different results

Conflict?	R(A)	W(A)
R(A)	NO	YES
W(A)	YES	YES
R(B)	NO	NO
W(B)	NO	NO

Conflict Serializability

def: possible to swap non-conflicting operations to derive a serial schedule.

∀conflicting operations O1 of T1, O2 of T2

- O1 always before O2 in the schedule or
- O2 always before O1 in the schedule

Operation Oi is a read or write of an object

Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

Conflicts

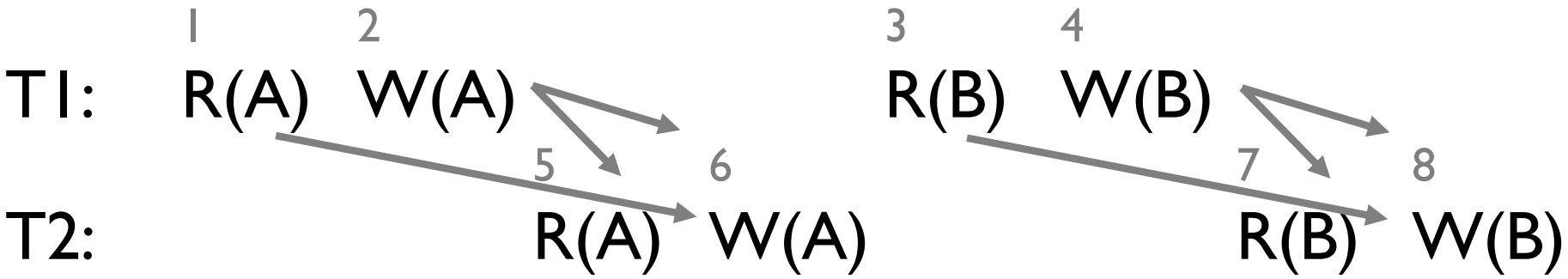
1,6 2,5 2,6 3,8 4,7 4,8

Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)

	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

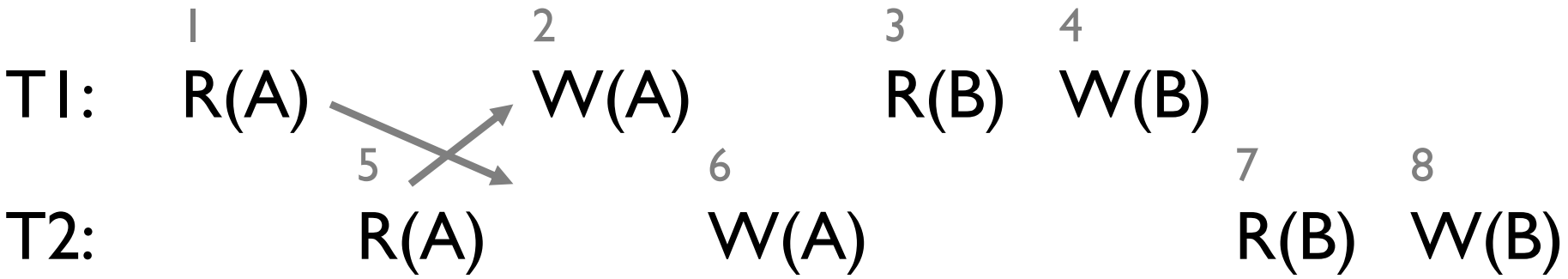
Conflict Serializable



Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

Not Conflict Serializable



Conflict Serializability

Transaction Precedence Graph

- Nodes are transactions
- Edge Ti → Tj if:

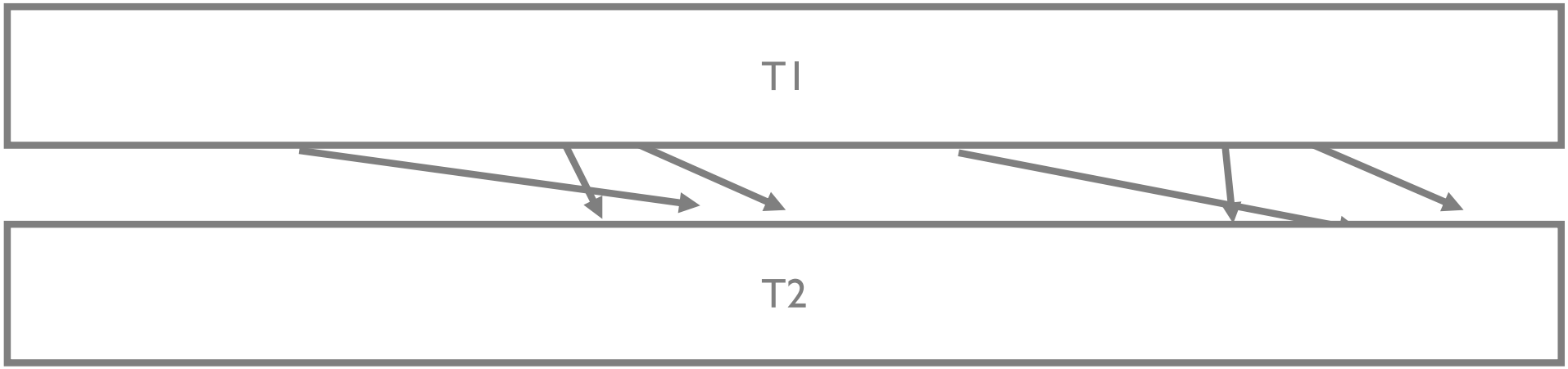
1. Ti read/write A before Tj writes A or
2. Ti writes some A before Tj reads A

Acyclic graph (no cycles) = conflict serializable!

Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

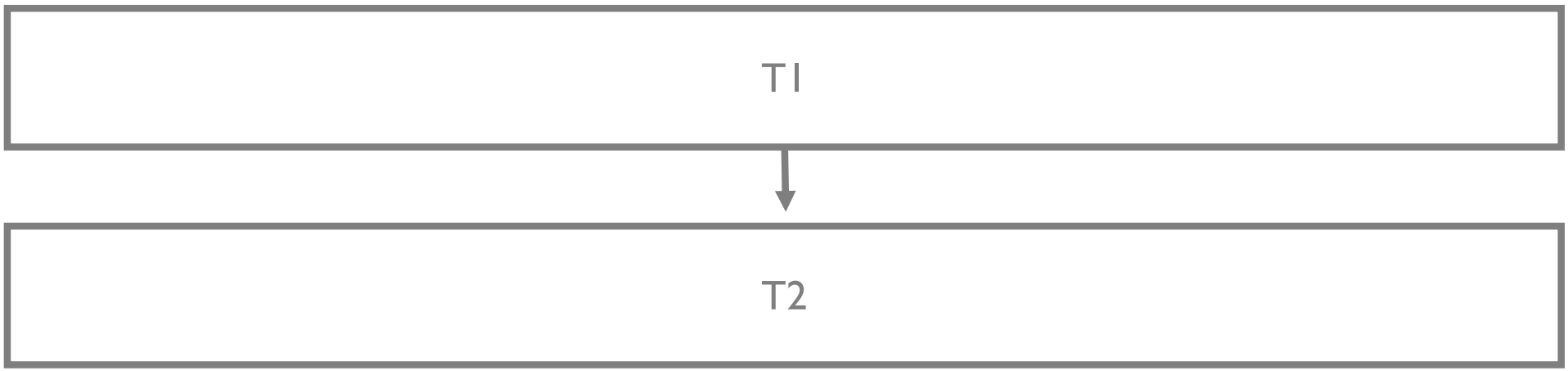
Conflict Serializable



Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

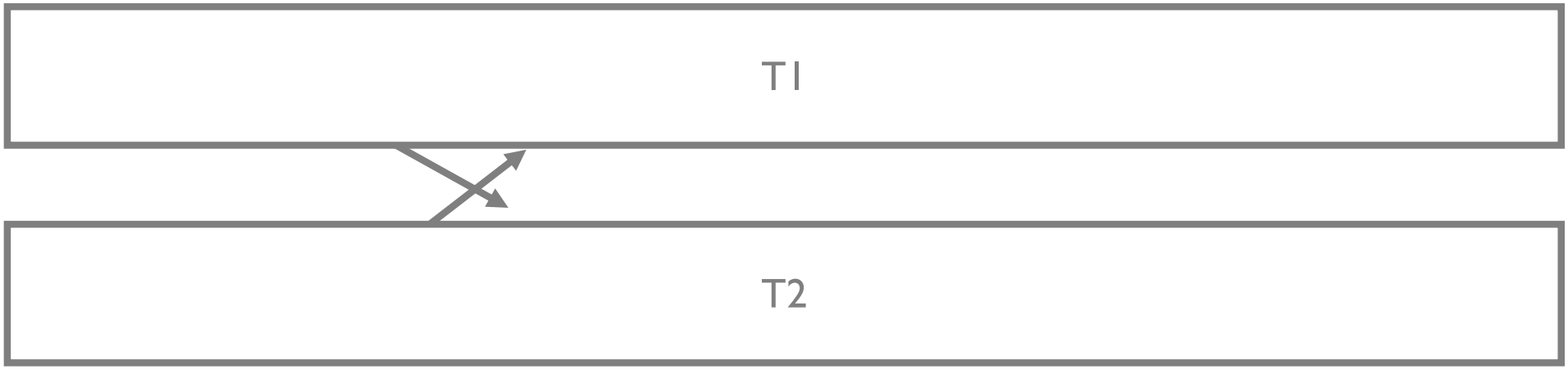
Conflict Serializable



Logical

	1	2	3	4
T1:	R(A)	W(A)	R(B)	W(B)
	5	6	7	8
T2:	R(A)	W(A)	R(B)	W(B)

Not Conflict Serializable



Commits/Aborts Complicate Things

So far, focused on schedule equivalence assuming that all transactions will commit.

But some transactions may abort and want to undo the changes.

These are OK right?

T1	R(A)	W(A)				R(B)
T2			R(A)			

T1	R(A)	W(B)	W(A)			
T2				R(A)	W(A)	

Fine, but what about COMMITing?

T1 R(A) W(A) R(B) ABORT

T2 R(A) COMMIT

Not recoverable

Promised T2 everything is OK. IT WAS A LIE.

T1 R(A) W(B) W(A) ABORT

T2 R(A) W(A)

Cascading Rollback.

T2 read uncommitted data → T1's abort undos T1's ops & T2's

Lock-based Concurrency Control

Everything so far has been definitions.

Want a *procedure* that will guarantee serializable schedules

Naïve approach:

- Lock database when starting xact
- Unlock database when xact ends (after COMMIT/ABORT)

Want something similar, but by locking objects (like records)

Lock-based Concurrency Control

Must get **S**hared(read) or e**X**clusive(write) lock BEFORE op

If other xact has lock, can acquire if lock table says so

Allowed?	T1 S	T1 X
T2 S	Y	N
T2 X	N	N

Can this schedule happen?

```
T1 R(A)  W(A)                    R(B) ABORT
T2                 R(A) COMMIT
```

YES

didn't discuss when to release locks

Lock-based Concurrency Control

Two-phase locking (2PL)

Growing phase: acquire locks

Shrinking phase: release locks

T1 R(A) W(B) W(A) ABORT

T2 R(A) W(A)

Uh Oh, same problem

Lock-based Concurrency Control

Strict two-phase locking (Strict 2PL)

Growing phase: acquire locks

Shrinking phase: release locks

Hold onto locks until commit/abort

Why? Which problem does it prevent?

T1	R(A) W(B)	W(A)		ABORT
T2			R(A) W(A)	

Guarantees serializable schedules! Avoids cascading rollbacks!

Review

Issues

- WR: dirty reads
- RW: unrepeatable reads
- WW: lost writes

Schedules

- Equivalence
- Serial
- Serializable

Serializability

- Conflict serializability
- how to detect

Conflict Serializable Issues

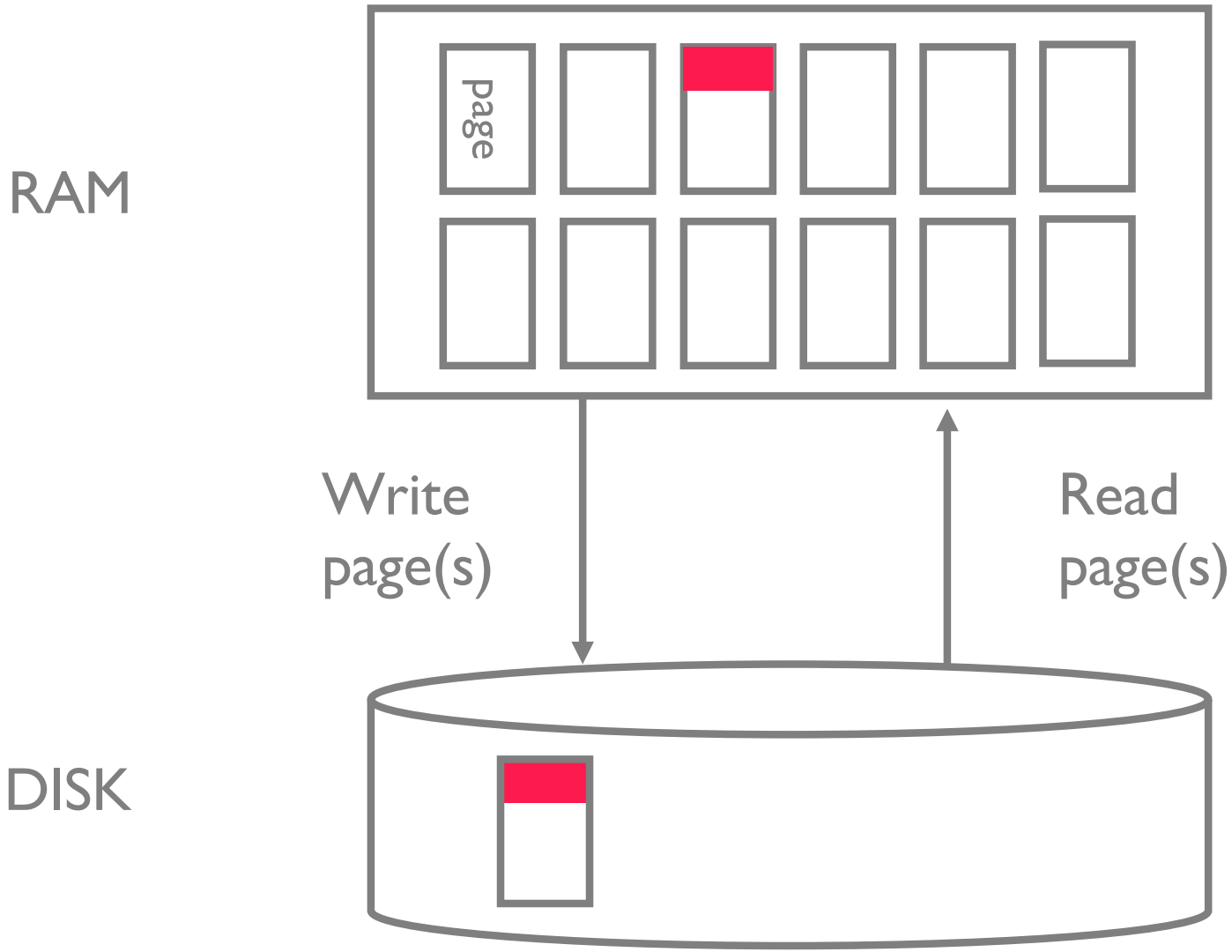
- Not recoverable
- Cascading Rollback

Strict 2 phase locking (2PL)

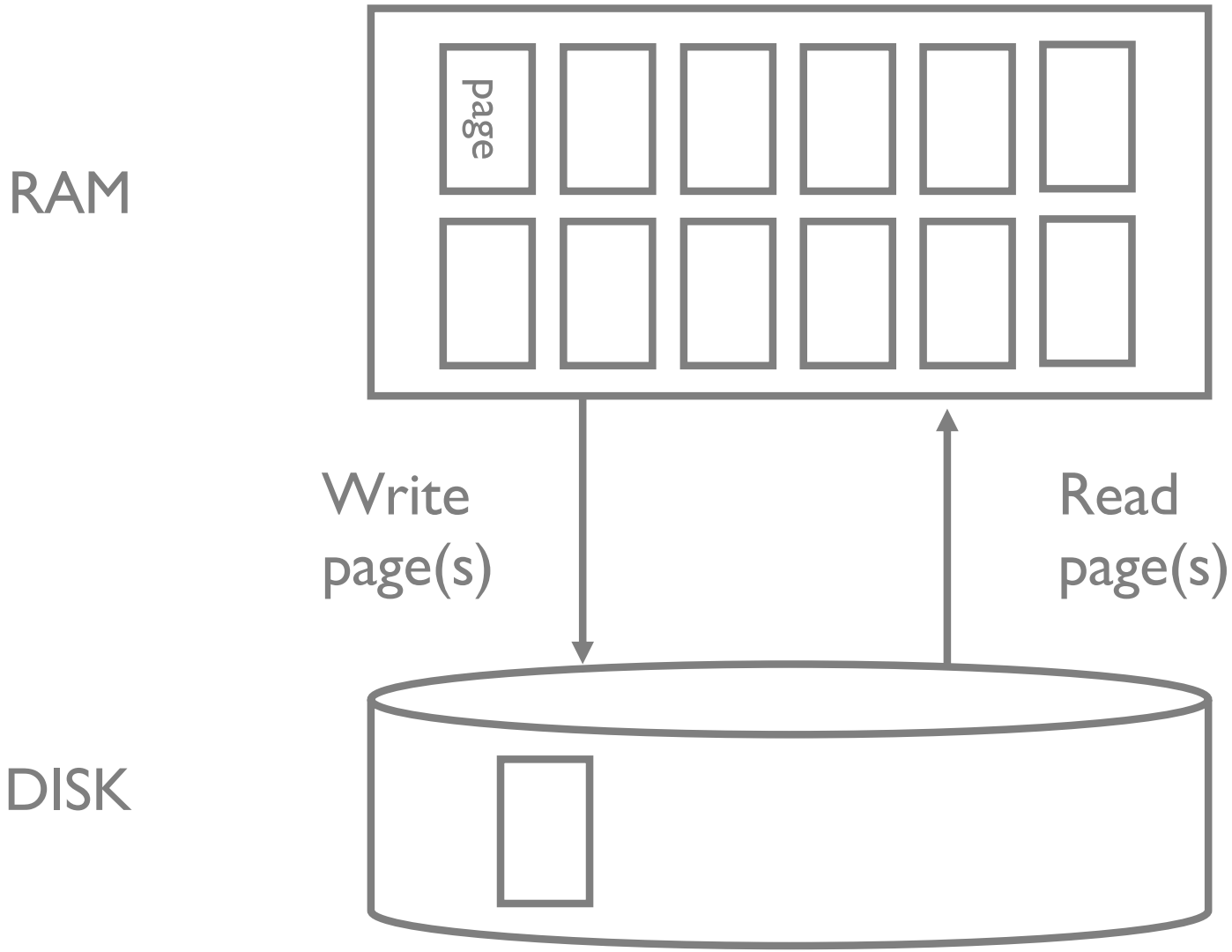
CRASH

Normal Execution

After a Crash

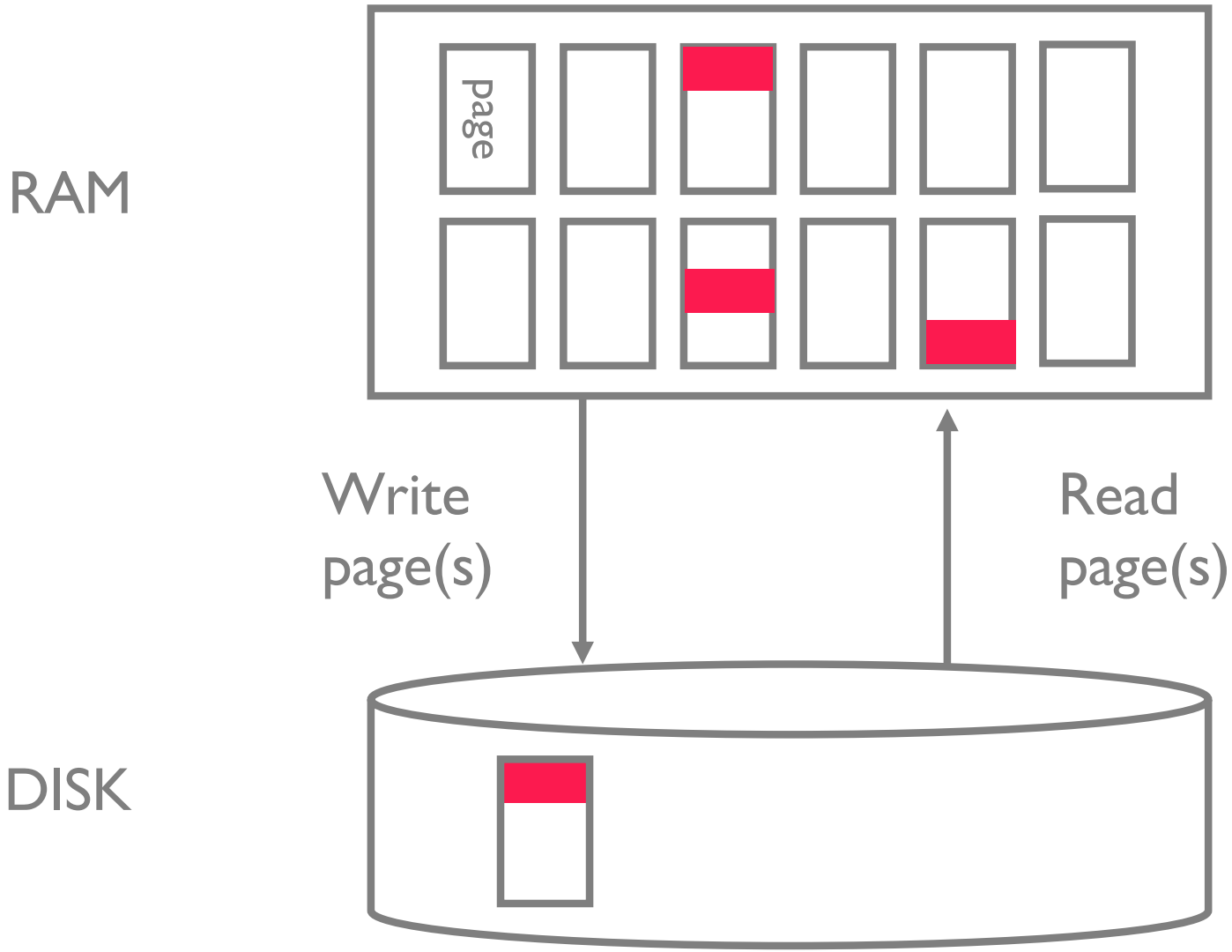


If DB did not say "OK, committed" State should look like:



If T1 Committed and DB said "COMMITED"

State should look like:



Recovery

Two properties: Atomicity, Durability

Assumption in class

- Disk is safe. Memory is not.
- Running strict-2PL

Need to account for

- when pages are modified
- when pages are flushed to disk

There's no _perfect_ recovery, just trade-offs

Recovery

Deal with 2 cases

When could uncommitted data appear after crash?

wrote modified pages before commit

If T2 commits, what could make it not durable?

didn't write all changed pages to disk

Aborts and Undos

If Tx aborts, must undo all its actions

- Ty that read Tx's writes must be aborted (cascading abort)
- Strict 2PL avoids cascading aborts

Use a log to know what actions to undo

1. A = 1
2. B = 5
3. C = 10
4. BEGIN T5
5. A = 10
6. B = B + A
7. C = B – 2
8. ABORT
9. undo 7
10. undo 6

...

Aborts and Undos

If Tx aborts, must undo all its actions

- Ty that read Tx's writes must be aborted (cascading abort)
- Strict 2PL avoids cascading aborts

Use a log to know what actions to undo

On crash, abort all non-committed xacts

1. A = 1
2. B = 5
3. C = 10
4. BEGIN T5
5. A = 10
6. B = B + A
7. C = B – 2
8. CRASH
9. undo 7
10. undo 6

...

Log

Log is the *ground truth*

Log records contain

- writes: old & new value
- commit/abort actions
- xact id & xact's previous log record

Initial protocol attempt:

Persist log records (write to disk) *before* data pages persisted

Is this enough?

Durability

Is there an execution that
writes log records before data pages
but is incorrect?
(e.g., not ACID)

Durability

Baseline scenario

- T1 writes to A in memory
- log record of write written to disk
- start writing page with A to disk…
- T1 commits

Durability

OK scenario

- T1 writes to A in memory
- log record of write written to disk
- start writing page with A to disk…
- *crash*
- T1 commits

Durability

OK scenario

T1 writes to A in memory

log record of write written to disk

crash

start writing page with A to disk…

T1 commits

Durability

Bad scenario

T1 writes to A in memory

T1 commits

log record of write is written to disk

start writing page with A to disk…

crash

Can undo help us?

No, need to *redo* T1, otherwise no durability!

Durability

Worse scenario

T1 writes to A in memory

T1 commits

crash

log record of write is written to disk

start writing page with A to disk…

Can undo help us?

Can't redo T1, no durability! Shareholders mad

Logs

Log is the *ground truth*

Log records contain

- writes: old & new value
- commit/abort actions
- xact id & xact's previous log record

Write ahead logging (WAL)

(1) Persist log records (write to disk) *before* data pages persisted (for UNDO)

(2) Persist all log records *before* commit (for REDO)

(3) Log is *ordered*, if record flushed, all previous records must be flushed

Note: DBMS can flush to disk or log at anytime as long as it follows WAL protocol

Aries Recovery Algorithm

Given WAL, 3 phases to recover

1. Analyze the log to find status of all xacts

   Committed or in flight?

2. Redo xacts that were committed

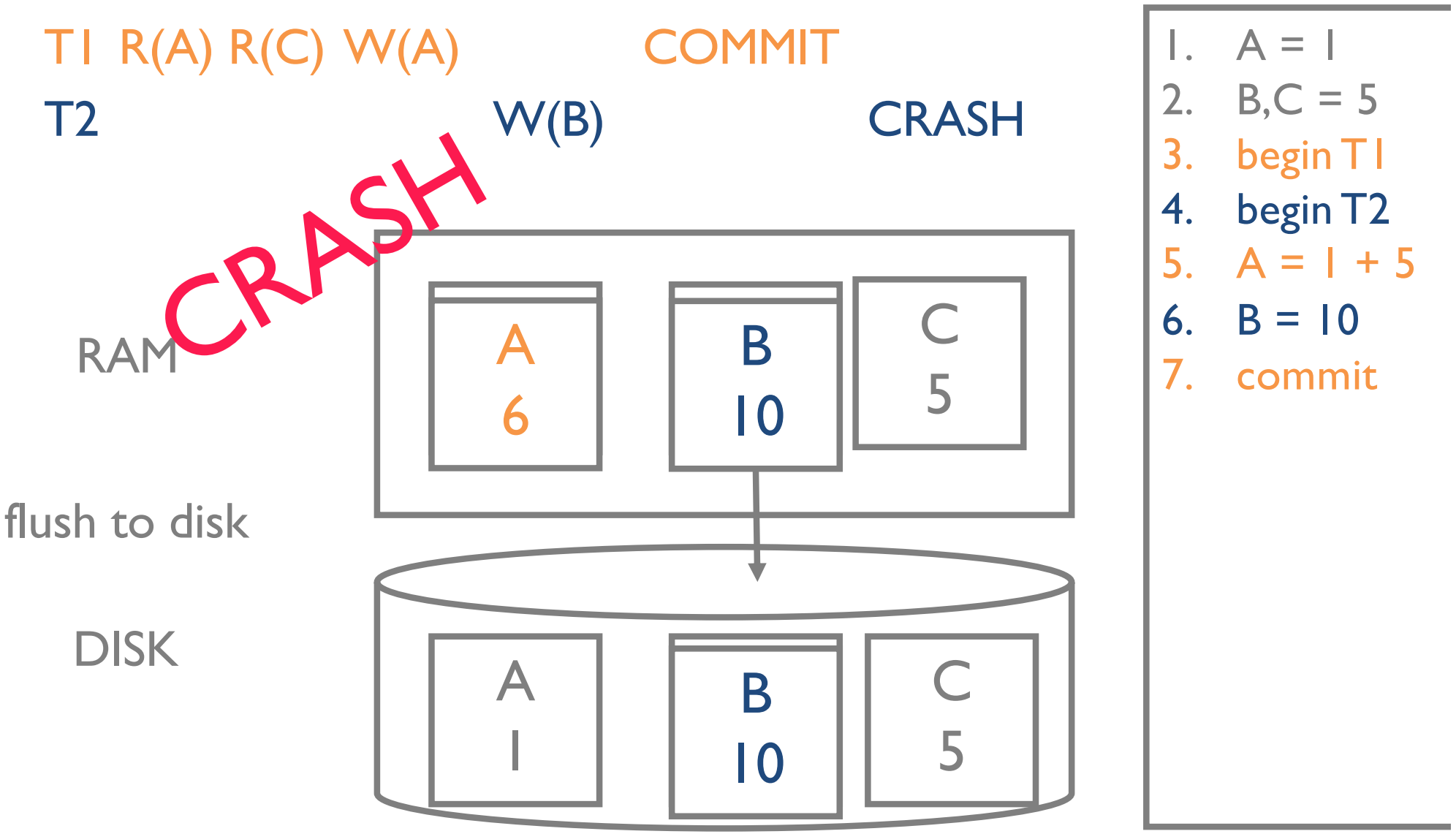
   Now at the same state at the point of the crash

3. Undo partial (in flight) xacts

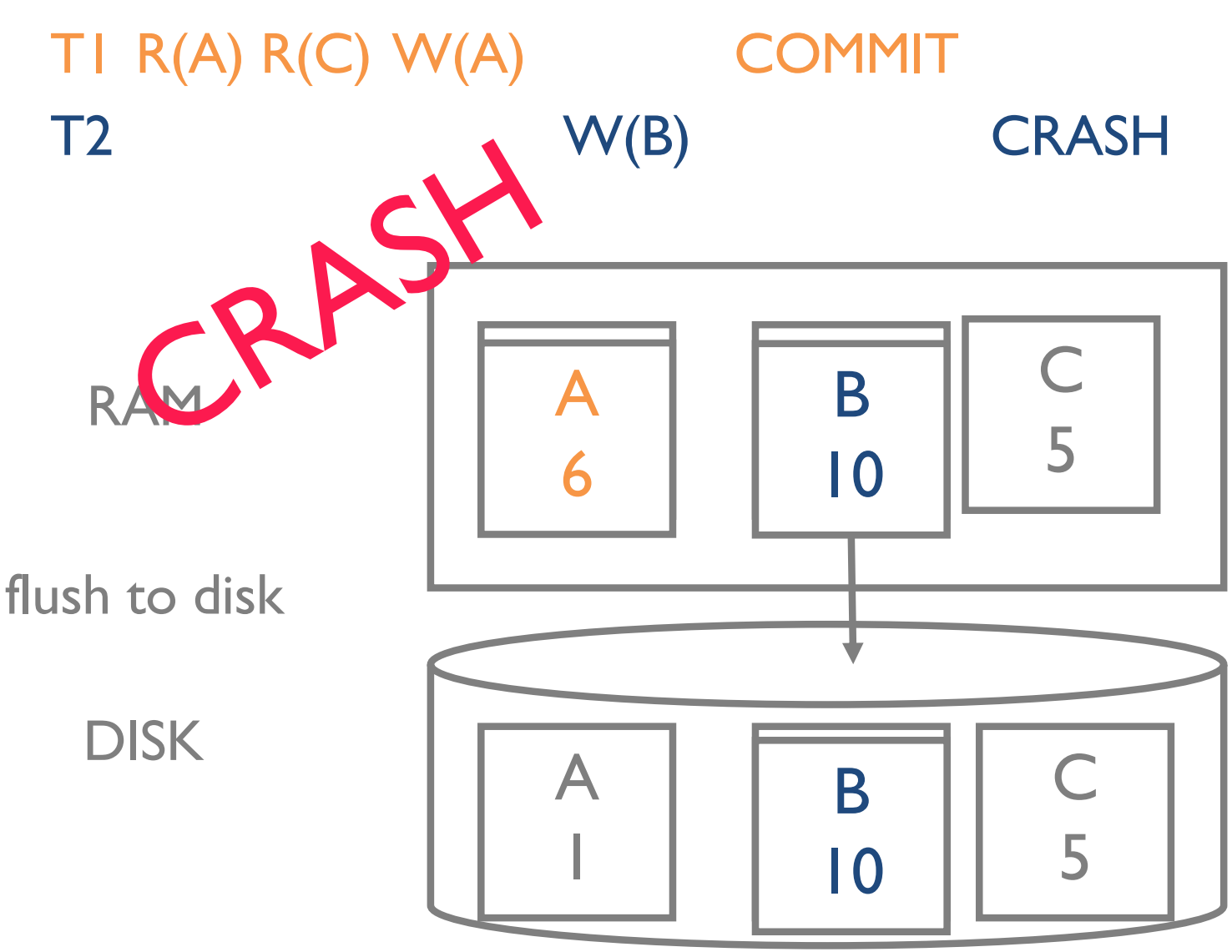
Note: redo/undo phases also follow WAL

Recovery is *extremely* tricky and *must be correct*

Aries (watch the lecture video for animation)



Aries: alternative flushing order

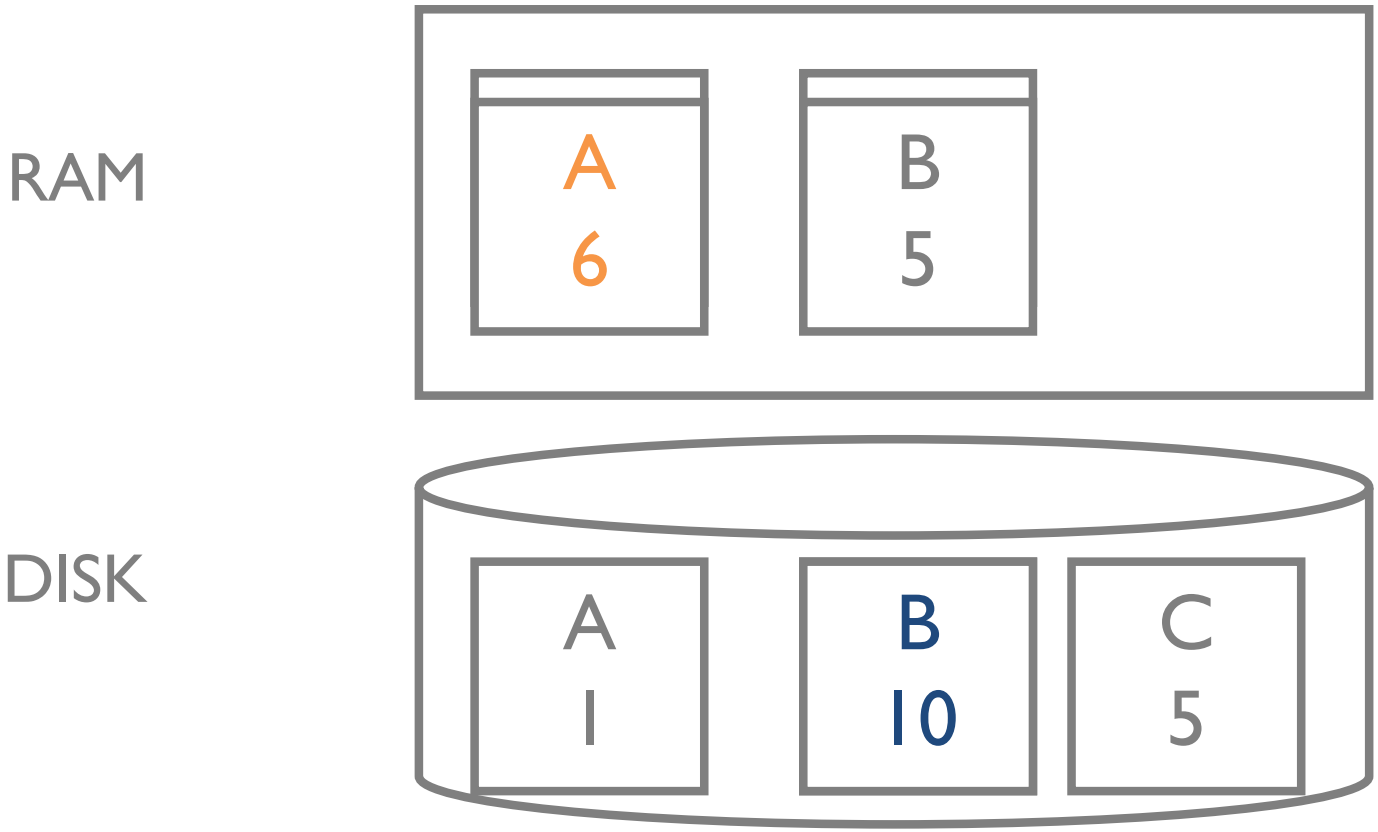


1. A = 1
2. B = 5
3. begin T1
4. begin T2
5. A = 1 + 5
6. B = 10
7. commit T1

Recovery (1)

Disk contains effects of op 6.

Analysis says: Redo 5, Undo 6



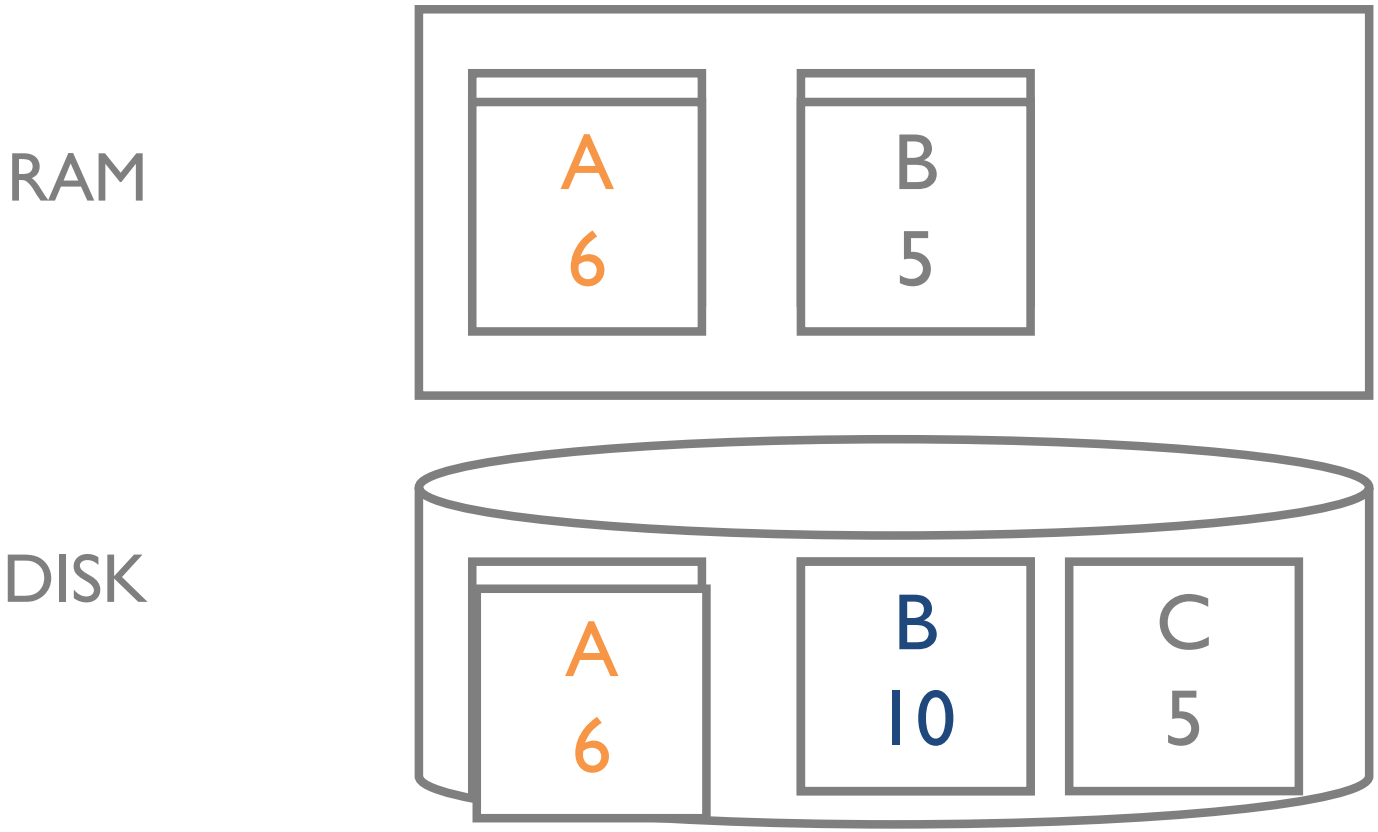
1. A = 1
2. B,C = 5
3. begin T1
4. begin T2
5. A = 1 + 5
6. B = 10
7. commit T1

Recovery(2)

Disk contains effects of op 6.

Analysis says: Redo 5, Undo 6

When flushing A=6, need to write log record

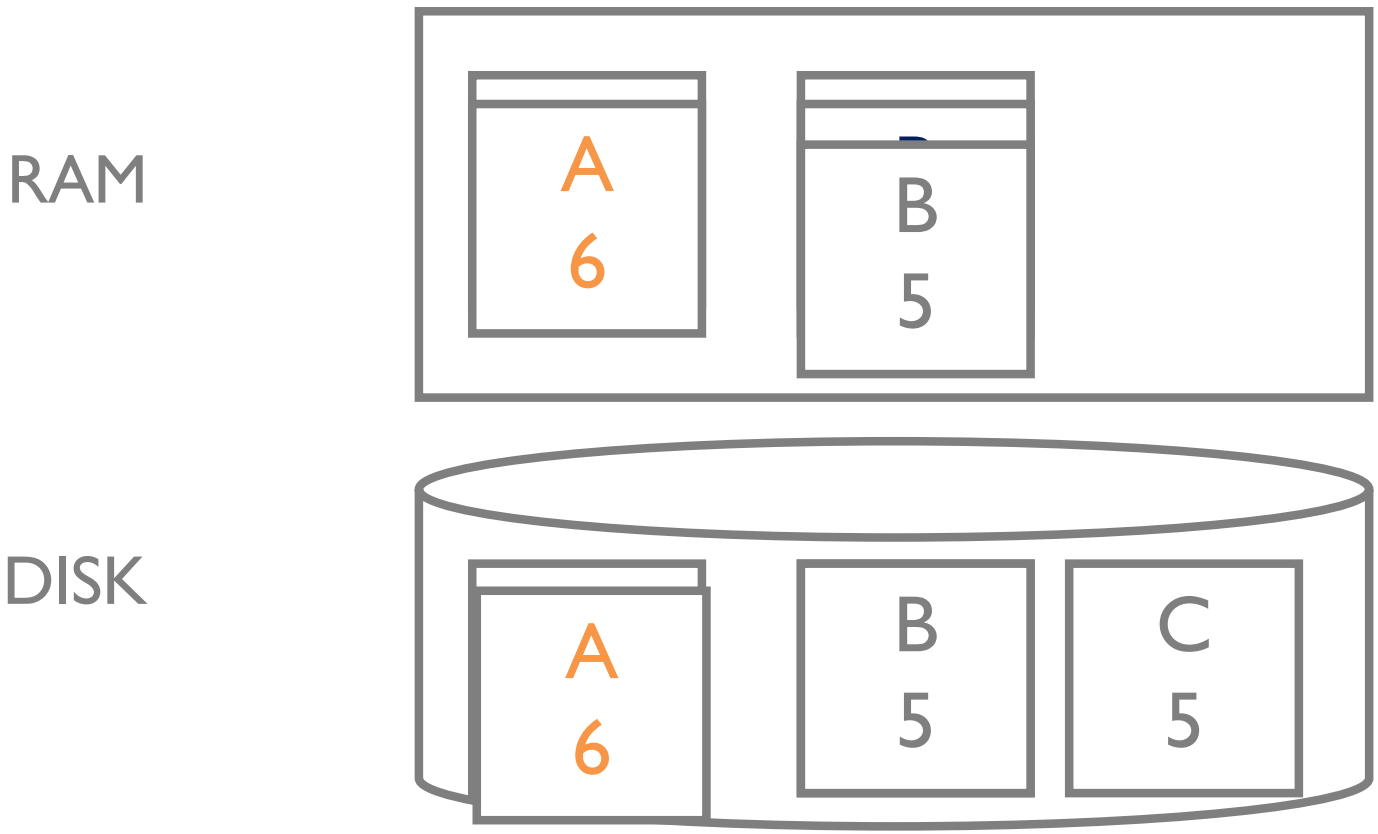


1. A = 1
2. B = 5
3. begin T1
4. begin T2
5. A = 1 + 5
6. B = 10
7. commit T1
8. redo op5
9. undo op6

Recovery(3)

Disk contains no modifications from T1 or T2.

Analysis says: Redo 5 and 6, Undo 6



1. A = 1
2. B = 5
3. begin T1
4. begin T2
5. A = 1 + 5
6. B = 10
7. commit T1
8. redo op5
9. redo op6
10. undo op6

Summary

Recovery depends on what failures are tolerable

Buffer pool can write RAM pages to disk any time

Recover to the moment of the crash, then undo all non-committed operations

WAL protocol

Recovery Manager ensures durability and atomicity via redo and undo

You should know

What transactions/schedules/serializable are

Can identify conflict serializable schedules

Can identify schedule anomalies

Can identify strict 2PL executions

Understand WAL and what it provides

Given an executed schedule, and a log file, run the proper sequence of undo/redos